Foosball Coding: Correcting Shift Errors and Bit Flip Errors in 3D Racetrack Memory

Samantha Archer, Georgios Mappouras, Robert Calderbank, Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
{samanta.archer, georgios.mappouras, robert.calderbank, sorin}@duke.edu

***Abstract*—Racetrack memory is a promising new non-volatile memory technology, especially because of the density of its 3D implementation. However, for 3D racetrack to reach its potential, certain reliability issues must be overcome. Prior work used per-track encoding to tolerate the shift errors that are unique to racetrack, but no solutions existed for tolerating both shift errors and bit flip errors. We introduce Foosball Coding, which combines per-track coding for shift errors with a novel across-track coding for bit flips. Moreover, our per-track coding scheme methodically explores the design of inter-codeword delimiters and introduces the novel concept of multi-purpose delimiters, in which the existence of multiple delimiter options can be used to provide additional information.**

Keywords—Racetrack memory; fault tolerance; error coding; endurance coding

I. Introduction

Racetrack memory [1, 2, 3, 4], also known as domain wall memory, is an exciting new memory technology that has recently been demonstrated in the lab. In racetrack memory, which we explain more fully in Section II, bits are stored on magnetic strips (i.e., the tracks) as the presence or absence of magnetic domains. The bits on a given track are shifted over one or more read/write ports, leading to a bit-serial access mechanism.

Racetrack memory offers the potential for far greater storage density than other technologies. Table 1 shows how racetrack compares to SRAM and DRAM, in terms of the effective cell size F^2 [2]. In particular, the density is outstanding for the 3D variant of racetrack memory, in which each track is in a U-shape with the bottom of the "U" positioned over a single read/write port on the substrate. Because of its vast density superiority, we focus here on 3D racetrack instead of the 2D variant in which tracks lay flat on the substrate.

To unlock the potential of racetrack memory for use in real systems, we must overcome its reliability challenges. Uniquely among current and emerging memory technologies, racetrack memory is susceptible to *shift errors*. As bits are shifted along the racetrack, it is possible for under-shifting ("insertion" in coding terms) and over-shifting ("deletion") to occur. As has been shown before, traditional error correcting codes, like Hamming codes, are unable to tolerate shift errors [5, 6, 7].[1]

Table 1. Effective cell size (F^2) for memory technologies

SRAM	DRAM	2D-Racetrack	3D-Racetrack
120 F^2	6-12 F^2	1-2 F^2	0.125-0.5 F^2

Prior work [5, 6, 7, 8] has addressed the shift error problem, including GreenFlag [7], which uses a combination of Varshamov-Tenegolts coding [9] and delimiters to solve the problem for 3D racetrack. However, none of this work is capable of also tolerating even a single bit flip error if the read/write head mis-reads a magnetic domain, either due to an error on the track or in the read/write head itself.

In this paper, we introduce Foosball Coding, a coding scheme that simultaneously tolerates both shift errors and bit flip errors.[2] Foosball Coding uses GreenFlag-like coding on a horizontal (per-track) basis, and it uses a vertical (across-track) Hamming code to correct bit flips. That is, Foosball Coding adds tracks that hold the Hamming code bits. This use of Hamming codes requires innovation because the Hamming bits themselves must tolerate shift errors, yet Hamming codes are not inherently resilient to shift errors.

Foosball Coding also introduces a new coding technique we call *multi-purpose delimiters* (MPD). GreenFlag, and other prior work [10, 11], uses delimiters between codewords. In the case of GreenFlag, the delimiters serve to detect whether a shift error is an insertion or deletion. We have observed that multiple delimiters could serve the same purpose. For example, there are multiple 6-bit delimiters that can detect a single insertion or deletion. Thus, we could choose the delimiters in a way that encodes additional useful information. In general, if we have D distinct delimiters, all of which serve the same purpose, then we have $log_2 D$ bits of data that we can exploit. In Foosball Coding, we use MPD to provide an optional added degree of error detection.

This work makes the following contributions:

- We present Foosball Coding, which is the first coding scheme to simultaneously tolerate shift errors and bit flip errors in 3D (and 2D) racetrack memory.
- We develop a systematic, constraint-based methodology for constructing delimiters that identify shift errors in the presence of a given error model.
- We introduce multi-purpose delimiters and show how to exploit them to increase the error detection capability of Foosball Coding.

[1] Consider a simple example. Imagine an n-bit codeword of alternating 0s and 1s, e.g., 010101. A single shift error would be equivalent to n or n-1 bit flips, which Hamming cannot tolerate.

[2] A foosball player can either be shifted or flipped.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00049

II. RACETRACK BACKGROUND

We now provide a short background on racetrack memory, focusing on what is most pertinent for Foosball Coding.

A. *Physical Implementation*

The basic components of racetrack memory are the magnetic strip (track), and the read/write ports that sit upon a substrate as seen in Figure 1. The track stores information in terms of magnetic regions, known either as magnetic domains (MDs) [1, 2, 3, 4] or magnetic objects called skyrmions [12, 13, 14], depending on the underlying technology used to create the memory. For the purpose of this paper, the two technologies are equivalent and thus we assume MDs for the rest of the paper.

Two different geometries have been proposed; a U-shaped 3D track (Figure 1a) and a line-shaped 2D track (Figure 1b). There are two main differences between 2D and 3D Racetrack memory. The first difference is that 2D Racetrack can have multiple read/write ports per track as the track is always adjacent to the substrate, whereas 3D Racetrack allows for only a single read/write port at the bottom of the U-shaped track. 3D Racetrack is projected to achieve significantly greater density because more tracks can be packed per area unit due to its geometry. Although this paper focuses on 3D Racetrack memory, Foosball can be also be applied to 2D Racetrack. For the rest of this paper, we assume 3D Racetrack with a single read/write port. However, for simplicity of illustration, our figures will use a 2D representation.

B. *Operation*

Regardless of the physical geometry of the Racetrack memory, a track can store a number of physical bit locations (logical 1 or 0). To read or write a bit, MDs must be shifted over the read/write port. By inducing current that runs parallel with the track, MDs can be shifted along the track. The track and the ports are physically fixed and cannot move. A port can access the bit location above it by either sensing if an MD is present (i.e., read) or inducing an MD (i.e., write). To read (write) multiple bits, consecutive read (write) and shift operations must be performed as shown in Figure 2a-c.

C. *Error Model*

Shift errors. Shift errors are the distinctive challenge with racetrack memory. Similar to prior work [5, 6, 7, 8], we model single shift errors with a probability of an insertion or deletion on every bit shift. A single shift error—caused by injecting too much or too little current along the track—causes the bits to be misaligned with respect to the read/write port by one bit position. Although it is theoretically possible to have a double shift error during a single shift operation (i.e., misalignment by two bit positions), prior work has shown that this is very unlikely [5] and thus this case is not considered. However, it is possible to have multiple single shift errors while performing multiple shift operations, resulting in a total of two (or more) shift errors while reading a block of data. For example, we could experience the (subtly tricky) situation in which an insertion and a deletion occur while reading a single codeword.

Flip errors. Bit flip errors have received less attention for two reasons. First, the unique challenge of shift errors has monopolized the interest of researchers. Second, there is no empirical data, to our knowledge, that characterize the likelihood of bit flips. Nonetheless, bit flip errors can occur when data is transferred through a memory channel from the memory controller to the physical memory and back. Furthermore, an implementation that protects against shift errors only would miss-categorize bit flip errors as shift errors. Thus, attempting to correct detected errors would actually cause misalignment between the bit positions and the read/write port and data corruption. For these reasons, attempts to address this error model have been made for 2D racetrack [15]. Similarly, we expect bit flip errors in 3D racetrack memory—in the bits themselves, or in the read/write head or memory channel—and thus we seek to protect 3D racetrack accordingly.

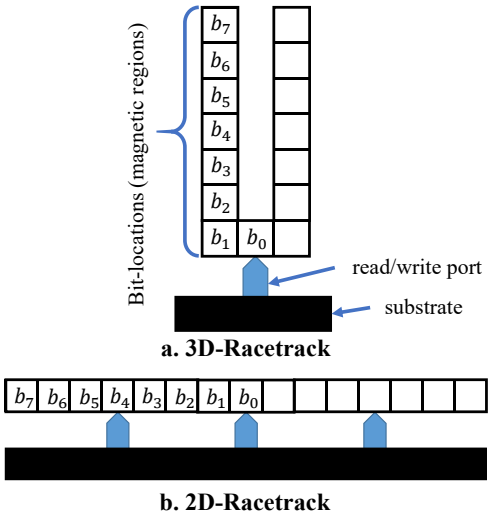

Figure 1. (a) 3D-Racetrack (a) and (b) 2D-Racetrack.

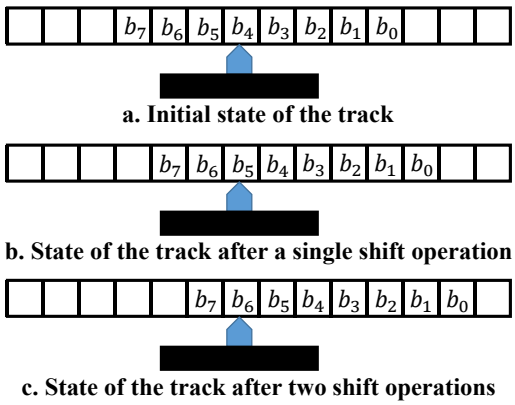

Figure 2. A system representation of 3D-Racetrack (a) in its initial state, (b) after a shift operation, (c) after two shift operations.

III. Prior Work

While there is a large body of work on racetrack memory itself and how to apply it to various levels of the memory and storage hierarchy [16, 17, 18, 19, 20], we focus here on fault tolerance. We divide the work into two sections, based on whether it targets 2D or 3D racetrack; all of the schemes for 2D racetrack rely on having multiple read/write ports, which is not feasible for 3D racetrack. Foosball Coding builds on the 3D shift error tolerance scheme called GreenFlag, by adding the ability to tolerate bit flips.

A. *Fault Tolerance for 2D Racetrack*

Chee et al. [8]. In this scheme, data is encoded using run-length limited codes. By reading the same data through multiple ports that have pre-determined distances between them, shift errors can be detected and corrected. Depending on the number of ports available, different fault tolerance levels can be achieved. No bit-flip protection is provided.

HiFi [5] and np-ECC [15]. HiFi adds delimiters to the edges of each track. Data bits are stored in the middle section of a track. At runtime, data bits and delimiter bits are read simultaneously via multiple ports. If the bit values of the delimiter bits differ from the expected values, a shift error is detected. HiFi can detect double shift errors and correct single shift errors. A subsequent extension of HiFi, called np-ECC, additionally provides single bit-flip correction for the delimiter (but not the data). However, it cannot provide fault tolerance guarantees in the case that both bit flips and shift errors are present simultaneously.

B. *Fault Tolerance for 3D Racetrack*

There are two prior fault tolerance schemes for 3D Racetrack, both of which focus on achieving shift error detection and correction with only a single read/write port. To our knowledge, no protection against bit-flips has been proposed.

Ollivier et al. [6]. Ollivier et al. [6] present a shift protection scheme that combines multiple coding and physical techniques. They propose a new physical way to access information on the Racetrack, called Traverse Read (TR). With TR, one can calculate the number of 1s in a track without performing shift operations. Additionally, delimiter bits are added to the edges of each track. Using TR, the weight (number of 1s) of the data and the delimiter bits can be calculated. This weight is called the signature of a track. Based on the current signature of a track, the signature after a shift operation can be predicted. If a mismatch occurs between the predicted value of the signature and the signature calculated after a shift operation, a shift error is detected. To guarantee that the current signature of a track can always be reliably recovered, they store it in STT-RAM, a memory technology that does not incur shift errors. While clever, this scheme has significant drawbacks. First, it can only detect and correct a single shift error for the entire track, limiting the practical length of each track. Furthermore, it provides no guarantees for bit-flip errors.

GreenFlag [7]. GreenFlag is a combination of (a) a code that can correct shift errors if it knows if the error is an insertion or a deletion, and (b) a delimiter that detects insertions and deletions.

The shift correction code is based on Varshamov-Tenegolts (VT) codes [9]. An n-bit VT codeword is constructed in two steps. First, the dataword bits are placed consecutively at the non-power-of-two positions in the codeword. Second, the power-of-two-positions are filled with bits that we refer to as "fill bits." We denote such a codeword as VT(n, k), where n is the size of the codeword and k the size of the dataword in bits. For example, VT(64,57) uses 57 dataword bits and 7 fill bits. The values of the fill bits are chosen to satisfy the following checksum equation:

$$\sum_{i=1\ to\ n} ic_i = 0, mod\ (n+1)$$

All valid VT codewords have a checksum equal to zero, and thus VT codes have easily implementable algorithms for correcting insertions and deletions, based on trying to make the checksum equal to zero again. However, VT codes must know whether the shift error is an insertion or deletion; there is no algorithm for correcting a shift error of unknown type.

Therefore, GreenFlag appends to each VT codeword a specific, predetermined delimiter that is chosen such that it can detect a desired number of insertions and deletions in the VT codeword.[3] For example, a 6-bit delimiter of 000111 can detect a single insertion or deletion in the preceding VT codeword. A 1-bit insertion will result in a delimiter of X00011, and a 1-bit deletion will result in a delimiter of 00111X (where X could be 0 or 1). Neither X00011 nor 00111X can be mistaken for a correct delimiter or for each other. Longer delimiters can be used to detect more insertions and deletions. We refer to the VT codeword with its subsequent delimiter as an *extended codeword*, and any number of extended codewords can be stored one after another in a track, depending on the length of the track.

At runtime, a read operation involves reading an entire extended codeword. While reading the VT codeword, the checksum is recalculated and compared to zero. Additionally, the delimiter bits are read are compared to predetermined values. If both the checksum and the delimiter bits are correct, then no shift errors have occurred. Otherwise, based on both the values of the delimiter and the checksum, respectively, the type of the shift error is decided and corrected. The correction is done in two steps. First, the bits are shifted accordingly to realign them with respect to the read/write port. Second, the correct VT codeword is reconstructed and the dataword bits are extracted from the non-power-of-2 positions.

[3] We defer for now the possibility of shift errors in the delimiter.

GreenFlag can detect single and double shift errors, and it can correct single shift errors with the VT code alone; it can correct double shift errors by realigning the read/write port and re-reading. However, GreenFlag cannot tolerate bit flip errors. It can be shown that a single bit flip error combined with a single shift error can cause a silent data corruption. This limitation motivates Foosball Coding.

IV. Delimiter Construction & Multi-Purpose Delimiters

Delimiters are a useful mechanism for detecting insertions and deletions—and, in fact, have been used in other contexts, including denoting the boundaries between codewords [10, 11]. Typically, a delimiter is a predetermined string of bits that is easily distinguishable.

For example, GreenFlag uses a 6-bit delimiter 111000 that can be used to detect deletion or insertion errors in the preceding VT codeword. A single insertion error can be detected as a one-position right shift in each bit with an added bit X at the beginning of the delimiter string, i.e., X11100. Similarly, a deletion error can be detected as a one-position left shift in each bit with an added bit X at the end of the delimiter string, i.e., 11000X.

However, this particular delimiter fails to distinguish shift errors if bit flips are introduced in the error model. For example, consider the case that a bit flip occurs in the 4th bit of the delimiter. This results in the delimiter string 111100, which is equivalent to the delimiter we get after a single insertion (X11100). Because different errors could result in the same delimiter string, shifts and bit flips cannot be distinguished from each other with the delimiter 111000.

In this work, we present what is, to the best of our knowledge, the first formal approach for construction of delimiters for different error models. In addition, we introduce the idea of multi-purpose delimiters (MPD), in which we find a set of compatible delimiters for a given error model. With MPD, we can choose which delimiter within the set to use, and this choice can convey additional information.

A. *Delimiter Construction (for a single delimiter)*

The main idea behind delimiter construction is that we can set certain constraints on a delimiter string to guarantee that it works for a given error model. Fundamentally, these constraints ensure that the correct delimiter cannot be mistaken for a delimiter that has been modified by any error in the error model. While our delimiter construction methodology is general, we consider two error models in this section: shift errors only and bit flips with shift errors in the same track.

In Table 2, we summarize how the possible errors are handled, based on whether they occur in the delimiter or in the codeword that precedes the delimiter. This summary is independent of Foosball Coding.

Table 2. Handling errors in codeword and delimiter bits

	error in codeword	**error in delimiter**
shift error	Detected by delimiter. Correction, if any, varies by scheme	May be detected or corrected by delimiter
bit flip	*varies by scheme*	May be corrected by delimiter

Error Model 1: Shift Errors

Let us assume that we want to create a delimiter $\boldsymbol{d}$ that would work in the case of a single shift error; for now, we assume the shift error is in the codeword. Let $\boldsymbol{d} = [d_1, d_2, \ldots d_q]$ be a q-bit delimiter. If an insertion has occurred, then we instead read $\boldsymbol{d_{I_1}} = [X, d_1, \ldots d_{q-1}]$. Thus, in order to be able to detect a single insertion, $\boldsymbol{d}$ and $\boldsymbol{d_{I_1}}$ must always be different by at least one bit, i.e., $\boldsymbol{d}$ and $\boldsymbol{d_{I_1}}$ must have a *Hamming distance* of at least 1. Similarly, if a deletion error occurs, we would read $\boldsymbol{d_{D_1}} = [d_2, \ldots d_q, X]$. To be able to detect a deletion we thus need $\boldsymbol{d}$ and $\boldsymbol{d_{D_1}}$ to have a Hamming distance of at least 1. Furthermore, we need $\boldsymbol{d_{D_1}}$ and $\boldsymbol{d_{I_1}}$ to have a Hamming distance of at least 1 in order to be able to distinguish an insertion from a deletion. We summarize these constraints here:

$$Constraint\ 1a\text{: } Hamming\_Distance(\boldsymbol{d}, \boldsymbol{d_{I_1}}) \geq 1$$
$$Constraint\ 2a\text{: } Hamming\_Distance(\boldsymbol{d}, \boldsymbol{d_{D_1}}) \geq 1$$
$$Constraint\ 3a\text{: } Hamming\_Distance(\boldsymbol{d_{D_1}}, \boldsymbol{d_{I_1}}) \geq 1$$

If a q-bit sequence satisfies all three constraints, then it can be used as a delimiter. For our example, delimiter 001 satisfies all three constraints and thus, can be used to detect single shift errors.

This process can be extended to construct a delimiter that can detect any number of shift errors by simply adding more constraints. For example, if we want to consider two shift errors, then we add constraints for $\boldsymbol{d_{D_2}} = [d_3, \ldots d_q, X, X]$ and $\boldsymbol{d_{I_2}} = [X, X, d_1, \ldots d_{q-2}]$. The more constraints we add, the larger q must be in order to find such a delimiter. For example, to be able to detect two shift errors we need a delimiter that is at least 5-bits long (e.g., 00111).

We have deferred until now the issue of shift errors in the delimiter itself. Our delimiter construction methodology produces delimiters that can detect most shift errors in the delimiters, but fundamentally they cannot detect all of them. Certain shift errors are easily detected. Consider a deletion (insertion) in the first bit of the delimiter; this is equivalent to a deletion (insertion) in any of the preceding codeword bits. However, if a shift error occurs in the last few bits of the delimiter, there is no delimiter that can guarantee that the error will be detected because we cannot control the bit values of the following codeword. For example, consider the delimiter 00111; a deletion in the last bit cannot be guaranteed to be detected, because it is always possible that the first bit of the subsequent codeword is a 1.

In this paper, we take a conservative approach to shift errors in delimiters. If a shift error results in a delimiter we can recognize (e.g., correct delimiter, single shift delimiter, etc.), we identify the codeword accordingly. If not, we declare a detected uncorrectable error (DUE).

Error Model 2: Shift Errors and Bit Flip Errors

Now consider the case that we want to create a delimiter that can detect either one single shift error or one delimiter bit flip. That means that even if a bit in the delimiter $\boldsymbol{d}$ flips, we should still be able to distinguish it from the case of a single shift error. In this case, $\boldsymbol{d}$ and $\boldsymbol{d_{I_1}}$ must always differ by at least two bits. Similar is the case for $\boldsymbol{d}$ and $\boldsymbol{d_{D_1}}$. However, no increase in the Hamming distance of $\boldsymbol{d_{I_1}}$ and $\boldsymbol{d_{D_1}}$ is needed as we assume that either a shift error or a bit flip will occur, but never both. Thus, our constraints are now:

$$Constraint\ 1b\text{: } Hamming\_Distance\left(\boldsymbol{d}, \boldsymbol{d_{I_1}}\right) \geq 2$$
$$Constraint\ 2b\text{: } Hamming\_Distance\left(\boldsymbol{d}, \boldsymbol{d_{D_1}}\right) \geq 2$$
$$Constraint\ 3b\text{: } Hamming\_Distance\left(\boldsymbol{d_{D_1}}, \boldsymbol{d_{I_1}}\right) \geq 1$$

Again, this process can be extended to construct a delimiter that can detect any number of shift errors and/or any number of delimiter bit flip errors, by adjusting and adding constraints. For example, delimiter 1001010 can detect either two shift errors or one delimiter bit flip error. Delimiter 00011010 can detect two shift errors and one delimiter bit flip error even if both shift errors and delimiter bit flip errors are present at the same time.

B. *Multi-Purpose Delimiters (MPD)*

Observation: *For a given bit-length q, there may be multiple q-bit delimiters that satisfy the delimiter construction constraints, and there is an opportunity to exploit the extra information carried by the choice of delimiter.*

Let us return to the example of finding a delimiter that can detect a single shift error. We showed that delimiter 001 suffices for this error model, and it is clear that 110 is an equally useful delimiter for this error model. Theoretically, we could use both delimiters in order to encode an extra bit of information. For example, using delimiter 001 would encode '0' and using delimiter 110 would encode '1'. However, it is not immediately clear if 001 and 110 are *compatible delimiters*, i.e., if they can be used without being mistaken for each other in the presence of shift errors.

In general, we seek a set of 2^m compatible delimiters, where (a) all of the delimiters can detect the desired number of insertions or deletions, and (b) the delimiters are all compatible with each other, i.e., no delimiter can be mistaken for any other delimiter despite any error in our error model. Condition (a) is always satisfied by the construction constraints for a single delimiter. To also satisfy condition (b) we need to introduce additional constraints between the 2^m delimiters.

In our running example, let $\boldsymbol{d^A} = \left[\boldsymbol{d_1^A}, \boldsymbol{d_2^A}, \ldots, \boldsymbol{d_q^A}\right]$ and $\boldsymbol{d^B} = \left[\boldsymbol{d_1^B}, \boldsymbol{d_2^B}, \ldots, \boldsymbol{d_q^B}\right]$ be two different delimiters that both satisfy the constraints 1a-3a. To ensure that $\boldsymbol{d^A}$ and $\boldsymbol{d^B}$ are compatible, they cannot be confused after a single bit flip. Additionally, we must ensure that $\boldsymbol{d_{I_1}^A}$ cannot be confused with $\boldsymbol{d_{D_1}^B}$. Furthermore $\boldsymbol{d_{I_1}^A}$ must be different from $\boldsymbol{d^B}$ even if a bit flip error happens in $\boldsymbol{d^B}$. Similar are the cases for $\boldsymbol{d_{I_1}^B}$. In short, the following constraints must apply:

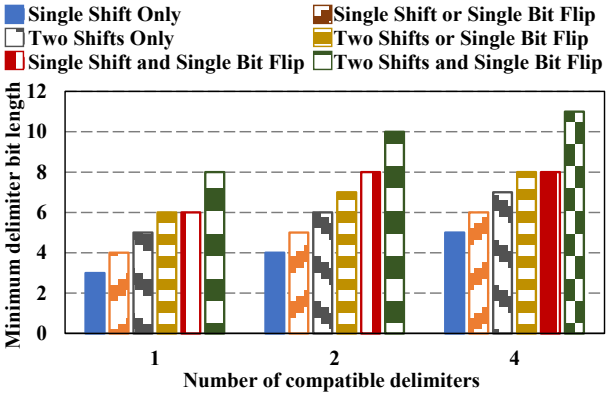

Figure 3. Minimum delimiter bit length (q) for different error models and different set size of compatible delimiters.

$$Constraint\ 1c\text{: } Hamming\_Distance\left(\boldsymbol{d^A}, \boldsymbol{d^B}\right) \geq 3$$
$$Constraint\ 2c\text{: } Hamming\_Distance\left(\boldsymbol{d_{D_1}^A}, \boldsymbol{d_{I_1}^B}\right) \geq 1$$
$$Constraint\ 3c\text{: } Hamming\_Distance\left(\boldsymbol{d_{I_1}^A}, \boldsymbol{d_{D_1}^B}\right) \geq 1$$
$$Constraint\ 4c\text{: } Hamming\_Distance\left(\boldsymbol{d_{I_1}^A}, \boldsymbol{d^B}\right) \geq 2$$
$$Constraint\ 5c\text{: } Hamming\_Distance\left(\boldsymbol{d_{D_1}^A}, \boldsymbol{d^B}\right) \geq 2$$
$$Constraint\ 6c\text{: } Hamming\_Distance\left(\boldsymbol{d_{I_1}^B}, \boldsymbol{d^A}\right) \geq 2$$
$$Constraint\ 7c\text{: } Hamming\_Distance\left(\boldsymbol{d_{D_1}^B}, \boldsymbol{d^A}\right) \geq 2$$

By enforcing constraints 1a-3a and constraints 1c-7c, we can find two delimiters that can tolerate single shift errors and be compatible. Using this methodology, and by adding constraints accordingly, we can find any set of 2^m compatible delimiters for any error model. For example, delimiters 0111101 and 1001010 are two compatible delimiters that can detect two shift errors or single bit flips. In Figure 3 we present the minimum bit length q that delimiters must have to find 1, 2, or 4 compatible delimiters for different error models.

Foosball Coding uses the delimiter construction methodology introduced in this section, including MPD.

V. Foosball Coding

We have developed two variants of Foosball Coding that provide different trade-offs between error tolerance and cost. We quantify cost as is typical in information theory, using a code's *rate*, defined as the ratio of dataword bits to codeword bits.

When we explain these codes, we consider the tracks to run parallel to each other in the horizontal direction (even though the tracks are actually 3D). Thus, we describe per-track coding as horizontal (or per-row) coding and across-track coding as vertical (or per-column) coding. We use the term *array* to describe the logical "rectangle" of bits from some number of rows and columns.

Our two variants of Foosball Coding are FC1 and FC2. FC1 (Section V.A) uses horizontal coding to provide error tolerance of either shift errors or bit flip errors (but not both at the same time). FC2 (Section V.B) uses horizontal and vertical coding to provide protection against shift errors and bit flip errors in a given array of bits.

A. *FC1: One Type of Error at a Time*

FC1 is the lower cost variant of Foosball Coding. Like GreenFlag, FC1 is a strictly horizontal code that encodes a dataword into a VT codeword with a delimiter. FC1 can tolerate 2 shift errors or 1 bit flip error per track. To achieve this, FC1 introduces two key innovations that provide low-cost protection against bit flip errors.

Innovation #1: Use MPD to encode Parity

FC1 uses MPD to generate two compatible delimiters that encode additional information. Specifically, with two distinct compatible delimiters, we have one bit of "free" information, and we use it to provide parity. Using our delimiter construction methodology for the given error model, we find that we need delimiters that are 7-bits long: 1001010 and 0111101. These delimiters are selected so that they can always be distinguished from each other, even if two shift errors or a bit flip error (but not both) occur per track.

Innovation #2: Use VT Checksum to Localize Bit Flip

FC1 uses insight into VT coding that allows one parity bit to detect *and correct* a single bit flip error. Recall that every VT codeword must satisfy a checksum equation over the n bits of the codeword (with numbering starting at position 1, not 0). Because the checksum is computed as a sum mod (n+1), a bit flip in position 1 and a bit flip in position n have the same effect on the checksum. A flip in position 1 can either add or subtract 1 (if 0→1 or 1→0, respectively), and a flip in position n can either subtract or add 1. Similarly, a flip in position 2 and position n-1 have the same effect of adding or subtracting 2. More generally, because of this symmetry, any discrepancy in the checksum due to a single bit flip can be localized to one of two bit positions.

Therefore, we know that the error is in one of two bit positions, and we know that one of the positions is on the left half of the codeword and the other is on the right half. By protecting one half of the codeword with parity—using the parity bit obtained via MPD—we can identify whether that half is the culprit. If it is not, we know that the other half is. Whichever half of the codeword is identified now uniquely identifies the bit position that is in error, and we correct it.

FC1 Encoding

The encoding process for FC1 is the following. A dataword is first encoded to a VT codeword as done in GreenFlag. Then, the left half bits of the VT codeword are logically XORed to produce a parity bit. If the value of the parity is 0, the VT codeword is appended with the delimiter 1001010. Otherwise, it is appended with the delimiter 0111101.

FC1 Decoding

As with GreenFlag, FC1 must read a whole extended codeword, calculate the checksum, and identify the value of the delimiter. If both the delimiter and the checksum are correct, we simply decode the VT codeword by reading the dataword bits from the non-power-of-two bit positions.

If the checksum is incorrect but the delimiter is correct, a single bit flip is detected. As explained earlier, we use the value of the checksum to isolate the bit flip error to just two bit positions in the VT codeword and then use the "free" parity given by the delimiter to correct it.

If both the checksum and the delimiter are incorrect, then a single shift error is detected and corrected using the VT decoding process in GreenFlag.

If the checksum is correct but the delimiter is incorrect, then either two shift errors or a single bit flip in the delimiter bits has occurred. Our delimiter construction methodology guarantees that both of our delimiters can detect and distinguish these different error cases. The shift errors are fixed by shifting bits accordingly to realign them with respect to the read/write port of the track. The flip error is corrected by simply re-writing the delimiter to its original value.

FC1 Error Model and Limitations

FC1 will immediately correct exactly one error—either shift error or bit flip error—per extended codeword. It can also detect 2 insertions or 2 deletions, in which case FC1 will realign the bit positions but will not retrieve the correct codeword. In this case, FC1 will report a detected uncorrectable error (DUE). We emphasize that, similar to GreenFlag, FC1 could attempt to re-read the codeword to retrieve the correct codeword bits. However, this last step is optional, and the memory controller or OS could decide how to handle these DUEs.

Any extended codeword that incurs other error scenarios—three or more shift errors, multiple bit flips, or a combination of shift errors and bit flips—is likely to lead to a silent data corruption (SDC).

Depending on the probabilities of shift errors and bit flip errors, as well as on the required level of error tolerance, FC1 may or may not suffice. We note that FC1 could be made more error tolerant—with stronger, but more costly, delimiters—or we could instead choose to use FC2.

B. *FC2: Both Types of Errors at Once*

FC2 is the more reliable, but more costly, variant of Foosball Coding. FC2 combines horizontal coding that is similar to GreenFlag with a vertical code that must overcome a key challenge with shift errors.

Horizontal Code

FC2 borrows GreenFlag for its horizontal code, where each track stores VT codewords that are made out of data bits (D) and fill bits (F). Each VT codeword is followed by delimiter bits (d). FC2 however, uses a *single* delimiter that differs from the two delimiters used in FC1. FC2's 8-bit delimiter 00011010 can distinguish between (a) up to two deletions or two insertions even in the presence of a single bit flip error in the delimiter, and (b) a single flip error in the delimiter bits.

FC2's horizontal code is used strictly to detect and correct shift errors in the VT codeword (and single bit flips in the delimiter bits). Unlike FC1, FC2 relies on a vertical code to

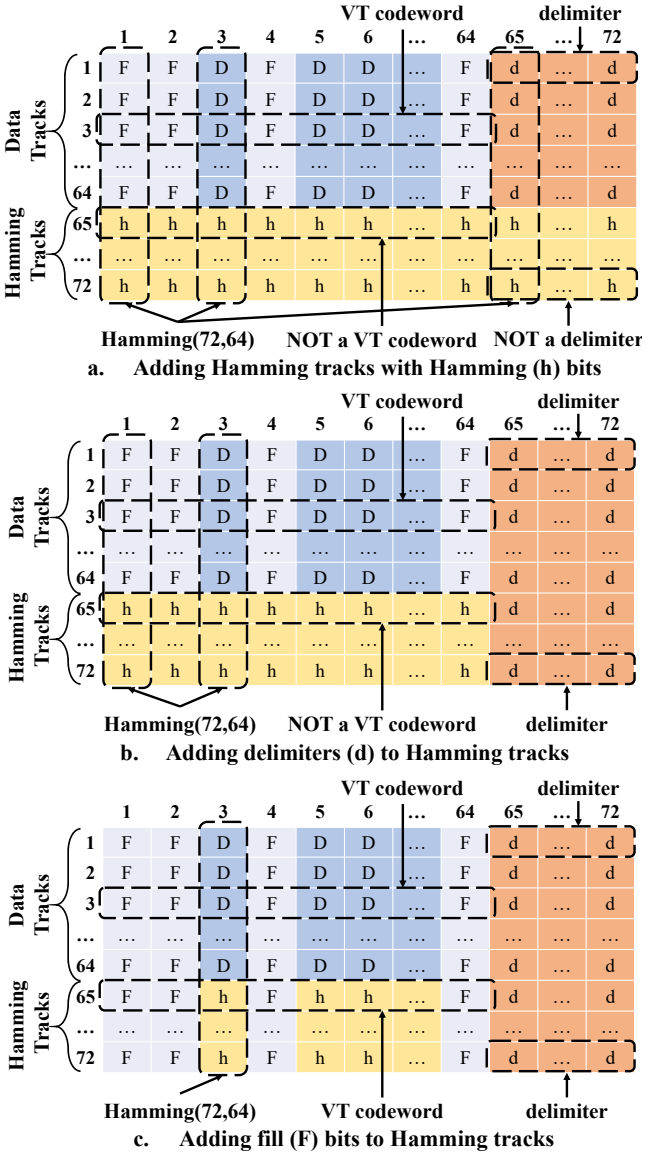

Figure 4. Transforming every track of the matrix to valid VT codewords through steps a, b, and c.

correct bit flips in the VT codeword (instead of FC1's use of parity stored with MPD).

Vertical Code

To provide bit flip protection, FC2 uses a vertical error correcting code (ECC), specifically a Hamming Code. Consider a (portion of) racetrack memory consisting of 64 tracks, where each track uses the horizontal code. Let us refer to these tracks as *data tracks*. We now add 8 Hamming Code tracks at the bottom, such that each column is now 64+8=72 bits long and forms a Hamming(72,64) codeword (see Figure 4a). Hamming(72,64) is a well-known code that provides single error correction and double error detection (SECDED). We refer to the added tracks as *Hamming tracks,* and each Hamming track stores Hamming parity bits (h).

The challenge with vertical coding—*and the reason that GreenFlag explicitly chose to use only horizontal coding*—is that the added Hamming tracks are seemingly unprotected from shift errors. The bits in the Hamming tracks do not form VT codewords but instead are a function of the bits in their respective columns. This implies that a single shift error in any of the Hamming tracks could render our vertical Hamming Code useless. We illustrate such an array in Figure 4a.

We overcome this problem by exploiting two observations. First, we observe that the delimiter bits in the data tracks do not need Hamming protection to detect and correct bit flips. Because our delimiter is explicitly constructed to be impervious to our error model, a single bit flip in the delimiter can be immediately detected and corrected, even if it is accompanied by a single shift error in the same delimiter bits. Thus, in the Hamming track bit positions that are in the same columns as delimiters in the data tracks, we can place any bit values we want instead of creating Hamming codewords. Specifically, we use those column positions of the Hamming tracks to store delimiters, as is done in the data tracks.

Thus, each data track consists of pairs of VT codewords and delimiters, and each Hamming track consists (for the moment) of pairs of "Hamming strings" (representing the Hamming Code bits for their respective columns) and delimiters. This new array is shown in Figure 4b.

Our second key observation is that we do not need to provide Hamming Code protection of the columns that correspond to fill bits in the data tracks. As explained in more detail later, the FC2 decoding process first performs horizontal decoding (using the fill bits) and then vertical decoding (during which the fill bits are irrelevant). Because the fill bits are not used after horizontal decoding, there is no need to correct them during vertical decoding and thus no need to protect them. Thus, in power-of-two positions in the Hamming strings, we can choose to place any bit values we see fit. Therefore, we choose fill bits that transform every Hamming string into a valid VT codeword, and we thus gain shift error protection for the Hamming tracks.

In Figure 4c, we present an example of this final FC2 array. In this example, there are 64 data tracks and 8 Hamming tracks that are all horizontally encoded as pairs of VT(64,57) codewords and delimiters (i.e., extended codewords). These extended codewords can detect and correct shift errors, and the vertical Hamming(72,64) can correct single bit errors and detect two bit errors.

We now describe in detail the encoding and decoding process of FC2.

FC2 Encoding

Encoding in FC2 is done at the array granularity. Before starting the encoding process, we arrange the input dataword bits in a temporary matrix with 64 rows, where each row represents a data track. In Figure 4, we explained how to encode by performing the horizontal encoding of each dataword, adding the vertical Hamming code, and then making the Hamming tracks resilient to shift errors. This encoding process is easiest to explain and we could have chosen to use it here, but in practice

it is easier to start with vertical encoding and then perform horizontal encoding.

Step 1 (Vertical Hamming Coding): We vertically encode all 64 rows of the matrix using Hamming(72,64). For each column, Hamming(72,64) produces 8 parity bits that are cached in 8 additional rows in the bottom of the temporary matrix (72 rows total). These additional rows correspond to the Hamming tracks.

Step 2 (Horizontal VT Coding): We horizontally encode all 72 rows of the matrix into VT codewords using the same encoding process as in GreenFlag, and the delimiter 00011010 is appended to each VT codeword.

After step 2, the entire temporary matrix (72 VT codewords with delimiters) is written to the racetrack memory, with each row written to a different track, comprising an FC2 array. All tracks can be written in parallel.

FC2 Decoding

The decoding process is also done at an array granularity. Thus, before decoding starts, all 72 tracks of a single array are read in parallel. The process can be separated into two steps.

Step 1 (Horizontal Decoding): Each track is individually, horizontally decoded with the VT/delimiter decoding algorithm described in GreenFlag. For each track, if shift errors are detected, the bit locations are shifted accordingly to re-align them with respect to the read/write port. If the error was a single shift error, the codeword bits are also corrected based on the VT decoding.

If two shift errors were detected (i.e., either two insertions or two deletions), we cannot use the VT code to correct the codeword bits.

If the two shift errors were one insertion and one deletion, then this combination is indistinguishable from a single bit flip error. As such, we rely on the vertical Hamming Code to subsequently correct any bit errors in such a row.

If a track has both a single shift error and a single bit flip error, the VT decoding process will still attempt to correct the codeword without being aware of the bit flip. Thus, the output of the VT decoding will (likely) be a wrong VT codeword. Again, such an error case can be later corrected (or at least detected) by the vertical Hamming Code. Regardless, the important thing is that the read/write port always gets re-aligned with respect to the bit locations in every track at the end of the first decoding step.

After we have fixed all shift errors and corrected each VT codeword (when possible), we simply extract the dataword bits from each VT codeword. For VT(64,57) we have 57 dataword bits. Thus, we now have 72 rows and 57 columns.

Step 2 (Vertical Decoding): We use Hamming(72,64) to vertically decode each column. The result is 64 rows of 57 columns. These are the final decoded dataword bits.

During the decoding of an array, we always go through step 1. However, step 2 is not always executed. If during step 1, two or more tracks report two shift errors, we declare a DUE and skip step 2. The reason we do this is that each track that has two shift errors is likely to extract multiple incorrect dataword bits from the incorrect VT codeword. Thus, there is a high probability that at least one column has two incorrect bit values, and an additional bit flip in the same column would lead to a SDC during the Hamming decoding in step 2. (Hamming(72,64) can only detect up to 2 errors). Thus, to avoid an SDC, we conservatively declare a DUE.

The ability of our vertical code to detect two errors per column is more powerful than it might appear at first. Consider the situation in which more than one row is mis-corrected in Step 1 (or one row is known to be uncorrectable and one or more other rows are mis-corrected). While Hamming(72,64) is likely to fail on multiple columns, it is extremely unlikely to fail on every column. If Hamming(72,64) finds a DUE on *any* column, FC2 declares a DUE for the entire array and avoids an SDC.

We emphasize that, similar to GreenFlag and FC1, when two shift errors are detected in a track, we could attempt to re-read its VT codeword (after realigning the bit positions with respect to the port). However, we consider this an optional step that the memory controller or OS could decide how to use.

FC2 Error Model and Limitations

For some error scenarios, FC2 guarantees no SDCs and/or no DUEs. For other error scenarios, FC2's error tolerance is a function of the locations of the shift and flip errors. In Table 3, we present 7 error scenarios. For each scenario, we show whether DUEs or SDCs are possible.

FC2 could be made more error tolerant, at additional cost, by choosing stronger horizontal and vertical codes. Longer delimiters and stronger ECC codes than Hamming(72,64) could tolerate more challenging error models and reduce the likelihood of SDCs and DUEs.

Table 3. Different Error Scenarios for FC2 and their possible outcomes

Error Scenario		FC2 Outcome
# of Shift Errors	# of Flip Errors	
1 per extended codeword	1 per FC2 array	no DUE no SDC
0 per FC2 array	1 per column of FC2 array	no DUE no SDC
2 in 1 extended codeword, ≤1 in all other extended codewords	0 per FC2 array	no DUE no SDC
1 per extended codeword	2 in different extended codewords	likely DUE no SDC
1 per FC2 array	1 per column of FC2 array	likely DUE no SDC
2 in 1 extended codeword, ≤1 in all other extended codewords	1 per FC2 array	likely DUE no SDC
1 per extended codeword	3 or more in different extended codewords	likely DUE or SDC

VI. Methodology

In this section, we describe how we evaluate FC1 and FC2, as well as compare it to GreenFlag [7]. For all three schemes, we use the horizontal code VT(64,57) that encodes 57 dataword bits to a 64-bit VT codeword.

We set the shift error probability to $P(s) = 10^{-6}$. We choose this value based on estimations from prior work [5]. However, to our knowledge, no experimental bit flip probabilities for Racetrack memory have been reported. Thus, we investigate a range of probabilities $P(f) = [10^{-6}, 10^{-9}]$.

A. *Analysis of FC1 and GreenFlag*

For GreenFlag and FC1, we build analytical, mathematical models that calculate the probability of a detectable uncorrectable error (DUE) and a silent data corruption (SDC) per VT codeword. Due to the simplicity of these schemes, we can enumerate the different scenarios where errors will be corrected or lead to DUEs or SDCs, and we can calculate their probabilities.

B. *Simulation and Analysis of FC2*

Unlike GreenFlag or FC1, FC2's error tolerance can depend on the location of the errors in the FC2 array. For example, consider the case of 3 bit flips and no shift errors in a single FC2 array. If all bit flip errors are in the same column, an SDC will occur. Otherwise, the errors could be corrected or cause a DUE. Due to the vast number of possible error location combinations, building an analytical model is practically impossible. Thus, to evaluate FC2, we rely on an in-house simulation.

Our FC2 simulator simulates the encoding and decoding processes in detail. Initially, for each array, it randomly generates 64 rows of datawords, each 57-bits long (64×57 matrix). All 57 columns are encoded using Hamming(72,64) to produce a 72×57 array. Each row is then encoded with VT(64,57) and the delimiter 00011010 is appended to produce the final 72×72 FC2 array.

Because error probabilities are quite low, naïve error injection would take a prohibitive amount of time. (The vast majority of simulated arrays would have zero errors.) Instead, we do the following.

Step 1: We analytically derive the probabilities of all non-negligible scenarios, including the no-error scenario. Each scenario is denoted with <*X=x, Y=y*>, where *x* is the number of shift errors and *y* is the number of bit flip errors in the FC2 array. We consider only the error scenarios with probability P(*X=x, Y=y*) ≥ 10^{-18} when *P(s)=P(f)*=10^{-6}. If an error scenario is less probable than that (e.g., 10^{-20}), considering it would not significantly change our results.

Step 2: We run a very large number (10^5) of simulations for each of the scenarios identified in the previous step. The simulator takes *x* and *y* as inputs. The simulator randomly picks *y* bits in the 72×72 matrix and flips their values, and then it randomly picks *x* bits to have a shift error. Each shift error has equal probability of being a deletion or an insertion. After all errors are induced, the simulator attempts to decode the FC2 array following the same process described in Section V.B. At the end of this decoding process, the simulator compares the decoded datawords with the original values and reports whether the result was correct data, a DUE, or an SDC.

Step 3: We weight the results from Step 2 using the probabilities from Step 1. Step 2 produces the probability of DUE and SDC given an error scenario *P(DUE | X=x, Y=y)* and *P(SDC | X=x, Y=y)*. We then analytically calculate the probability of each error scenario occurring in the FC2 matrix P(*X=x, Y=y*) based on the probabilities of *P(s)* and *P(f)*. By using the formula $P(DUE) = \sum_{x,y} P(DUE \mid X = x, Y = y) \times P(X = x, Y = y)$ we estimate the overall probability of DUE. Similar is the case for SDC.

C. *Primary Metric: DUEs and SDCs per Dataword Bit*

Because GreenFlag, FC1, and FC2 have different code rates and protect different numbers of dataword bits, it can be difficult to fairly compare them even after we have calculated *P(DUE)* and *P(SDC)* for each scheme.

To make the comparison fair, we study the expected number of DUEs and SDCs per dataword bit read. Let us use FC2 as an example and let us assume that it has a probability of SDC per FC2 array of $P_{FC2}(SDC)$. The expected number of FC2 arrays we must read to get a single SDC is then $E_{FC2}(SDC) = 1/P_{FC2}(SDC)$. However, for each FC2 array read, we access 57×64=3648 dataword bits. Thus, the total number of expected dataword bits we need to read to get a single SDC is:

$$E\left(\frac{datword\ bits}{SDC}\right) = E_{FC2}(SDC) \times 3648 = \frac{3648}{P_{FC2}(SDC)}$$

The expected SDCs per dataword bit is simply the reciprocal of the equation above. Similar is the case for the expected DUEs per dataword. The general form of the equation is:

$$E\left(\frac{DUEs}{dataword\ bit}\right) = \frac{P(DUE)}{\#\ dataword\ bits\ protected}$$

VII. Results

We now present our results for FC1 and FC2 and compare them to GreenFlag. We re-evaluate GreenFlag for our new error model that allows both bit flip and shift errors to occur.

A. *DUE and SDC Probabilities*

We start by presenting results for the probabilities of DUE and SDC for each code, as these results will help us better understand the expected number of DUEs and SDCs per dataword bit that we report next. In Figure 5, Figure 6, and Figure 7 we present the probabilities *P(DUE)* and *P*(*SDC*) for GreenFlag, FC1, and FC2 respectively. The x-axis is the value of P(f), and we remind the reader that P(s) is fixed to 10^{-6}. The y-axis is *P(DUE)* and *P*(*SDC*) in logarithmic scale. The probabilities for GreenFlag and FC1 are calculated for a single extended codeword (VT codeword + delimiter), whereas for FC2 they are for an entire FC2 array.

Our first observation is that *P*(*SDC*) for GreenFlag is very high because it can tolerate no bit flip errors, and its *P*(*SDC*) increases linearly with P(f). FC1 achieves 2-4 orders of

magnitude lower *P*(*SDC*) than GreenFlag, but P(DUE) for FC1 is as high as GreenFlag because they both declare DUEs for the same error scenario (i.e., two shift errors in an extended codeword).

FC2 can achieve significantly better *P*(*SDC*), with as low as 10^{-20} for P(f)=10^{-9}. However, *P*(*DUE*) for FC2 is higher than FC1 for P(f)>= 10^{-7}. This is because FC2 will report a DUE even if only 2 out of the 72 tracks in the FC2 array have two shift errors. On the contrary, FC1 can detect two shift errors for each track individually.

B. *DUEs and SDCs per dataword bit*

We now present the expected number of DUEs and SDCs per dataword bit read. In Figure 8 we compare the number of DUEs per dataword bit for GreenFlag, FC1, and FC2. We observe that GreenFlag and FC1 have the same number of DUEs per dataword bit as they provide the same *P(DUE)* while protecting the same number of dataword bits. However, we see that FC2 can provide up to 6 orders of magnitude fewer DUEs per dataword bit. Even in the cases where the P(DUE) for FC2 was higher than the P(DUE) of FC1, FC2 can still provide the same or fewer DUEs per dataword bit. That is because FC2 protects many more dataword bits than FC1 or GreenFlag.

However, the number of DUEs per dataword bit is relatively high even for FC2. For P(f)=10^{-9}, FC2 has roughly 10^{-16} expected DUEs per dataword bit. For a system that constantly accesses dataword bits at a bandwidth of 1GB/s, around 27 DUEs will be reported in a year. However, we note that the majority of these DUEs could be potentially corrected by just attempting to read the data again.

In Figure 9 we present the same results but now for SDCs. From Figure 9, we see that FC2 achieves far fewer SDCs per dataword bit, achieving as low as 10^{-23} SDCs per dataword bit. That means that, even if we access dataword bits with a rate of 1GB/s, we would only see 1 SDC every 1,000,000 years. The SDCs increase to 1 every 10,000 years, 10 years, and a few days for P(f)=10^{-8}, P(f)=10^{-7}, and P(f)=10^{-6}, respectively.

We conclude that GreenFlag and FC1 do not provide sufficient error tolerance when considering bit flip errors.

C. *Error Scenarios that Cause DUEs and SDCs with FC2*

We were interested to see which error scenarios cause the most DUEs and SDCs with FC2. This information will help us understand how we could increase the fault tolerance of FC2.

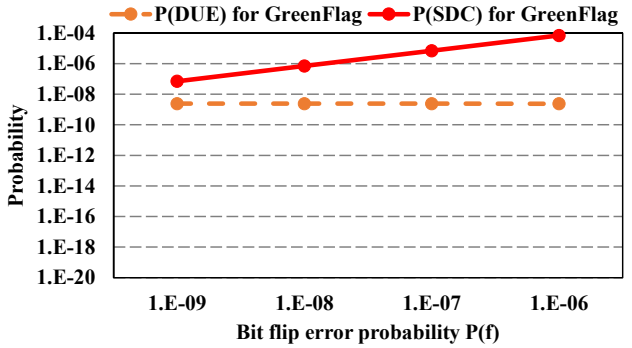

Figure 5. Probability of DUE and SDC for GreenFlag

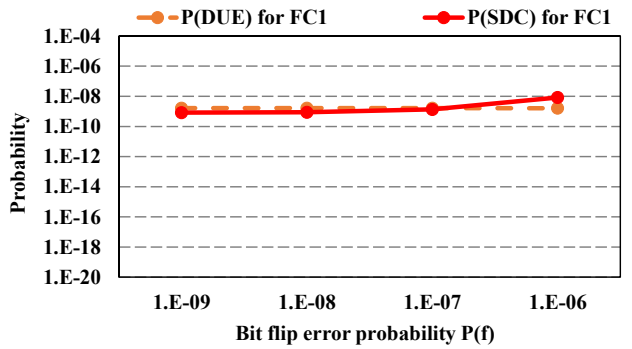

Figure 6. Probability of DUE and SDC for FC1

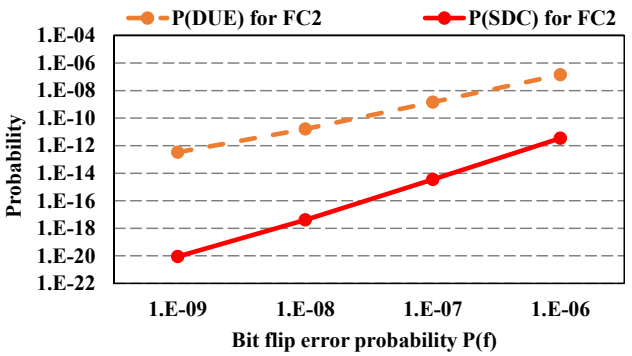

Figure 7. Probability of DUE and SDC for FC2

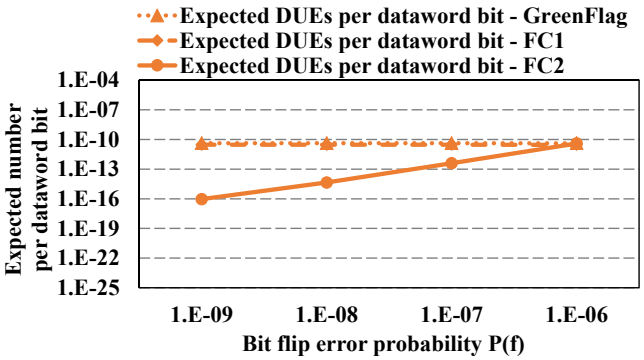

Figure 8. E(DUE) per dataword bit for GreenFlag, FC1, and FC2

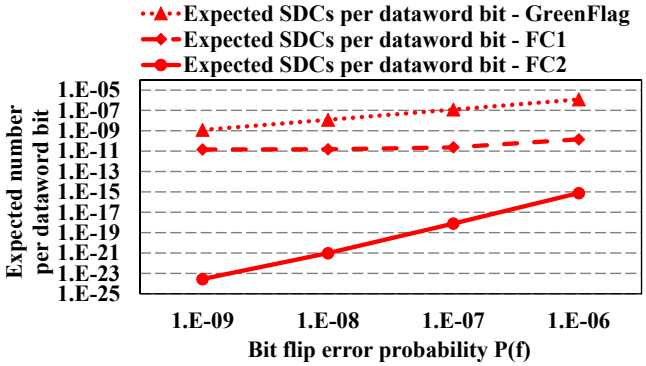

Figure 9. E(SDC) per dataword bit for GreenFlag, FC1, and FC2

In Figure 10 we present the percentage of DUEs caused by different error scenarios. We include the error scenarios with the two highest percentages of DUEs, as well as the percentage of DUEs for all other error scenarios. We observe that the error scenario that causes the highest percentage of DUEs is always responsible for more than 55% of them. Moreover, for $P(f) \geq 10^{-8}$ we see that the main error scenario that causes DUEs is multiple bit flips in the FC2 array. That indicates that, in order to reduce the number of DUEs, we should use a stronger ECC code. For example, we could use Hamming(7,4) instead of Hamming(72,64) to provide SECDED protection in every 7 tracks rather than every 72. However, that would come with a significant reduction in the overall rate of FC2. For $P(f) < 10^{-8}$, we observe that the main error scenario that causes DUEs is multiple shift errors. Thus, for such a scenario we could use smaller VT codes, like VT(16,11), to provide shift protection every 16 bits rather than every 64.

The results for SDCs are similar and are presented in Figure 11. The main difference is that SDCs depend more on flip errors. Even when $P(f)=10^{-9}$, the main error scenario that causes SDCs is 2 shifts with 2 flips, followed by no shifts and 3 flip errors.

D. *GreenFlag vs FC2 in the Absence of Bit Flip Errors*

Lastly, we compare GreenFlag against FC2 in the absence of bit flip errors. Although the main goal of this paper was to provide codes that can protect against both shift and bit flip errors, it is interesting to see how the two codes compare when we consider only shift errors.

Thus, we evaluate both GreenFlag and FC2 once more, assuming that P(f)=0. In Figure 12 we present the expected number of DUEs and SDCs per dataword bit for both codes. As we observe, FC2 can provide almost 7 orders of magnitude fewer DUEs per dataword bit and 8 orders of magnitude fewer SDCs per dataword bit. For a memory system that constantly accesses dataword bits with a rate of 1GB/s, GreenFlag (with VT(64,57)) would experience 1 SDC every few days, whereas FC2 would experience 1 SDC every 10^7 years. Thus, even in the case that only shift errors are considered, FC2 can provide significantly greater error tolerance than GreenFlag.

VIII. Conclusion

In this paper, we have introduced Foosball Coding, the first scheme for protecting 3D racetrack memory from shift errors and bit flip errors. We presented two variants, FC1 and FC2, that provide different trade-offs between cost and error tolerance. As part of the development of Foosball Coding, we developed a methodology for the construction of delimiters and we introduced the multi-purpose delimiters that can provide additional information. The experimental results show that both FC1 and FC2 successfully tolerate shift errors and bit flip errors. When racetrack memory makes the leap from the lab to production, Foosball Coding is poised to deal with its expected error models.

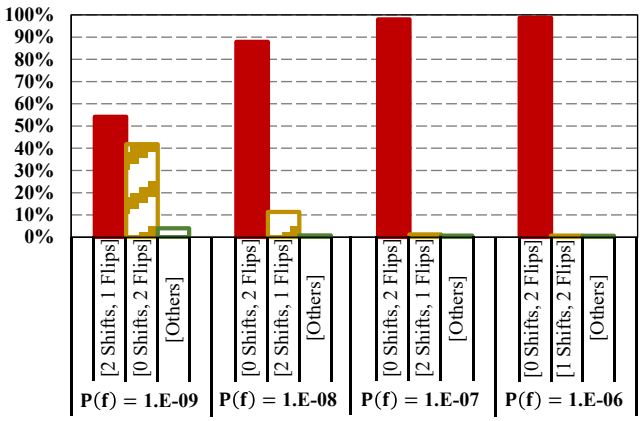

Figure 10. Error scenarios that cause the highest percentage of DUEs in FC2

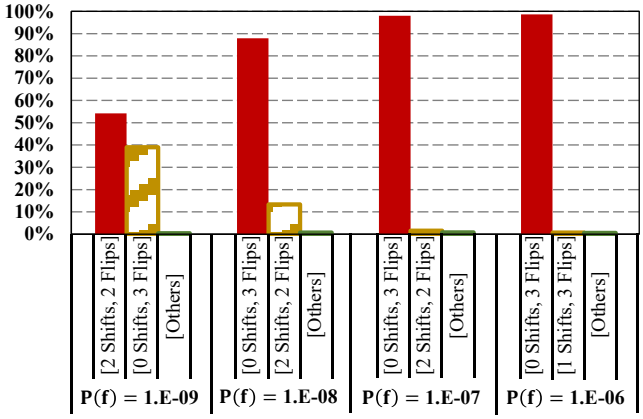

Figure 11. Error scenarios that cause the highest percentage of SDCs in FC2

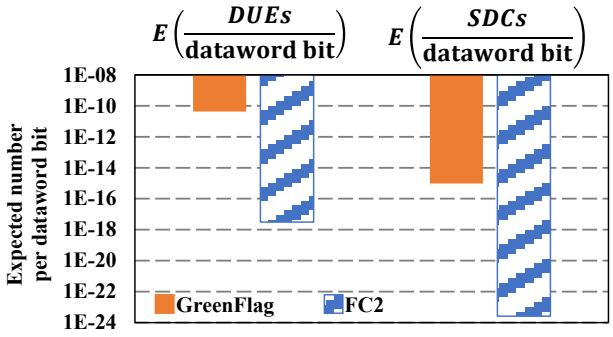

Figure 12. E(DUE) and E(SDC) per dataword bit for GreenFlag and FC2 when P(f)=0

Acknowledgments

This material is based on work supported by the National Science Foundation under grants CCF-142-1177 and CCF171-7602.

REFERENCES

[1] S. Parkin and S.-H. Yang, "Memory on the Racetrack," *Nature Nanotechnology*, vol. 10, no. 3, p. 195, 2015.

[2] S. S. Parkin, M. Hayashi, and L. Thomas, "Magnetic Domain-Wall Racetrack Memory," *Science*, vol. 320, no. 5873, pp. 190–194, 2008.

[3] W. Zhao, Y. Zhang, H. Trinh, J. Klein, C. Chappert, R. Mantovan, A. Lamperti, R. Cowburn, T. Trypiniotis, M. Klaui *et al.*, "Magnetic Domain-Wall Racetrack Memory for High Density and Fast Data Storage," in *International Conference on Solid-State and Integrated Circuit Technology*. IEEE, 2012, pp. 1–4.

[4] Y. P. Ivanov, A. Chuvilin, S. Lopatin, and J. Kosel, "Modulated Magnetic Nanowires for Controlling Domain Wall Motion: Toward 3D Magnetic Memories," *ACS nano*, vol. 10, no. 5, pp. 5326–5332, 2016.

[5] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu, "Hi-fi Playback: Tolerating Position Errors in Shift Operations of Racetrack Memory," in *SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 694–706.

[6] S. Ollivier, D. Kline, R. Kawsher, R. Melhem, S. Banja, and A. K. Jones, "Leveraging Transverse Reads to Correct Alignment Faults in Domain Wall Memories," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 375–387.

[7] G. Mappouras, A. Vahid, R. Calderbank, and D. J. Sorin, "GreenFlag: Protecting 3D-Racetrack Memory From Shift Errors," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 1–12.

[8] Y. M. Chee, H. M. Kiah, A. Vardy, E. Yaakobi *et al.*, "Coding for Racetrack Memories," *Transactions on Information Theory*, vol. 64, no. 11, pp. 7094–7112, 2018.

[9] R. Varshamov and G. Tenengolts, "Codes Which Correct Single Asymmetric Errors ," *Automatika i Telemkhanika*, vol. 161, no. 3, pp. 288–292, 1965.

[10] G. Tenengolts, "Nonbinary Codes, Correcting Single Deletion or Insertion (Corresp.)," *Transactions on Information Theory*, vol. 30, no. 5, pp. 766–769, 1984.

[11] F. Paluncic, K. A. Abdel-Ghaffar, and H. C. Ferreira, "Ínsertion/Deletion Detecting Codes and the Boundary Problem," *Transactions on Information Theory*, vol. 59, no. 9, pp. 5935–5943, 2013.

[12] R. Tomasello, E. Martinez, R. Zivieri, L. Torres, M. Carpentieri, and G. Finocchio, "A Strategy For The Design Of Skyrmion Racetrack Memories," *Scientific reports*, vol. 4, p. 6784, 2014.

[13] A. Fert, V. Cros, and J. Sampaio, "Skyrmions On The Track," *Nature nanotechnology*, vol. 8, no. 3, p. 152, 2013.

[14] Y. Huang, W. Kang, X. Zhang, Y. Zhou, and W. Zhao, "Magnetic Skyrmion-Based Synaptic Devices," *Nanotechnology*, vol. 28, no. 8, p. 08LT02, 2017.

[15] X. Wang, C. Zhang, X. Zhang, and G. Sun, "np-ECC: Nonadjacent Position Error Correction Code for Racetrack Memory," in *International Symposium on Nanoscale Architectures*. IEEE, 2016, pp. 23–24.

[16] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-Tape Architecture For GPGPU Cache Hierarchies," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 253–264.

[17] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "TapeCache: A High Density, Energy Efficient Cache Based On Domain Wall Memory," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2012, pp. 185–190.

[18] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative Modeling of Racetrack Memory, a Rradeoff Among Area, Performance, and Power," in *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 100–105.

[19] Z. Sun, W. Wu, and H. Li, "Cross-Layer Racetrack Memory Design for Ultra High Density and Low Power Consumption," in *Design Automation Conference*. IEEE, 2013, pp. 1–6.

[20] S. Mittal, "A Survey of Techniques for Architecting Processor Components Using Domain-Wall Memory," *Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 2, p. 29, 2017.